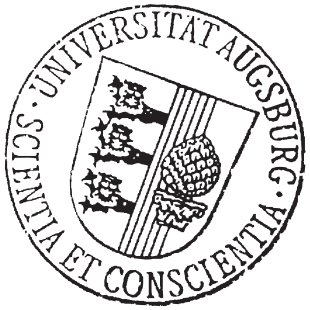
Universität Augsburg



ISPTAP – Instruction Scratchpad Timing Analysis Program: Features and Usage

Stefan Metzlaff

Report 2013-09 July 2013

Institut für Informatik
D-86135 Augsburg

Contents

Abstract

The Instruction Scratchpad Timing Analysis Program (ISPTAP) is a static timing analysis tool developed for the D–ISP (dynamic instruction scratchpad). It features the timing analysis of the CarCore and the ARM Cortex M0 processors and supports common instruction memories of embedded systems, like caches and scratchpads. In this report we describe the timing analysis tool including the timing models of the supported architectures. To validate the timing models we quantify the overestimation of the calculated WCETs compared to the measured execution times obtained from cycle-accurate processor simulators. Furthermore, we describe the usage and the configuration of the ISPTAP tool.

1 Introduction

The ISPTAP tool was developed during the work on the D–ISP (dynamic instruction scratchpad) [30] to quantify its impact on the WCET (worst-case execution time). Since its modular design, supporting multiple architectures and instruction memory types, it is also intended to be used for other purposes. Therefore, we will describe the architecture of the tool and give a short validation of the supported architectures. Currently, ISPTAP supports two processor architectures: The CarCore [35] (with the Infineon TriCore ISA [48]) and the ARM Cortex M0 [5] (the smallest processor implementing the ARMv6-M architecture [4]).

The Section 2 of the report describes the ISPTAP tool and is adopted from [30] extended by the timing model specification of the ARM Cortex M0 processor. Furthermore, in Section 3 we describe the usage of the tool and its configuration files. The Section 4 provides additional information regarding the licence and the availability of ISPTAP.

2 ISPTAP – A Static Timing Analysis Tool for the D–ISP

The *Instruction Scratchpad Timing Analysis Program* (ISPTAP) supports the program flow analysis, the pipeline analysis for the CarCore and the ARM Cortex M0 processor, and memory analysis for common instruction memories in embedded systems. For the description of the applied state-of-the-art low-level analysis techniques of the supported instruction memories and their extensions for the Dynamic Instruction Scratchpad (D–ISP [34, 33, 32]) refer to [30]. The proposed tool is targeted to the analysis of simple processor architectures (without branch prediction, prefetching, out-of-order execution, or any other speculation) and only provides a subset of common methods of static timing analysis that are useful for the instruction memories under observation. So the feature set of ISPTAP is much smaller than for commercial or research WCET tools like aiT [21] or OTAWA [7], which support e.g. data memory analysis including address analysis for data accesses, out-of-order execution, branch predictor analysis, loop bound determination, recursion, run-time optimisations (for large applications), and IDE integration. The ISPTAP tool was developed in C++ bearing in mind to easily replace or enhance parts/steps of the analysis and extend the supported memory types. An overview of the ISPTAP tool is depicted in Figure 1 showing the data flow of the WCET analysis. The classical division of a static analysis consists of the three steps: *program flow analysis*, *low-level analysis*, and *WCET calculation* as e.g. described in [14]. In the figure the low-level analysis is split into pipeline analysis and instruction memory analysis to show that these steps are separated and to highlight the main focus of the ISPTAP tool, which is the instruction memory analysis. These four steps performed by ISPTAP during static timing analysis are shortly described as:

- **Program Flow Analysis**: The executable of the program under analysis is split into basic blocks, connected in a control flow graph, and enriched with loop bounds to model the possible flow through the application.
- **Pipeline Execution Cost Analysis**: For each basic block the execution cost is determined without taking the memory system into account.

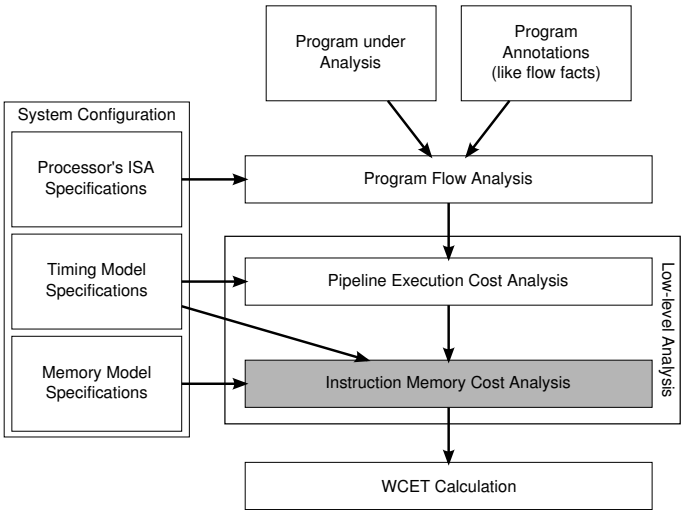


Figure 1: Analysis steps and data flow of ISPTAP

- **Instruction Memory Cost Analysis**: Depending on the memory type that is used in system configuration the cost caused by the instruction memory system is determined and added to the basic blocks as memory penalty.
- **WCET Calculation**: By finding the worst-case path of the application taking the cost of pipeline and memory for each basic block and the program structure into account the WCET estimation is calculated.

The analysis steps are described briefly in the following sections. Also a brief validation of the employed pipeline timing models is presented.

2.1 Program Parsing and Structural Representation

2.1.1 Executable Parsing

The application that is to be analysed has to be provided as linked executable, because it contains the addresses of data and instructions of the application as it will be run on the system. The knowledge of the addresses is of importance for the timing analysis, because the execution cost of a program is sensitive to the alignment of code and data. During this analysis step the application is split into basic blocks, which will be connected to a control flow graph per function. Calls to other functions are represented as nodes in the control flow graphs. ISPTAP also supports indirect jumps and calls by the help of user annotations that contain their possible target addresses.

2.1.2 Program Representation

To represent the structure of the entire application under observation that contains the entry function (e.g. the main function of the application) and all reachable called functions a so called *supergraph* [40] is created. The supergraph adds called functions as control flow graph only once and enhances them with call and return points depending on the calling context. So a function that is called multiple times from different contexts is represented only once, but it is entered from different call points for the different contexts. This reduces the overall size of the complete control flow graph and still allows the differentiation of multiple calling contexts in the following analysis steps.

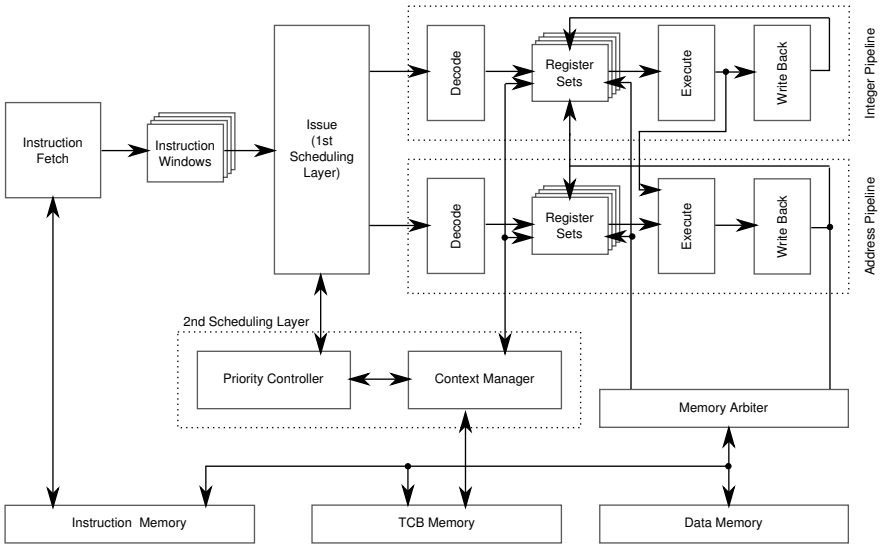


Figure 2: Block diagram of the CarCore processor (according to [35])

2.1.3 Loop Annotation

Loops within the application need to be provided with an upper bound otherwise the WCET cannot be calculated correctly. Thus the application programmer has to provide a file containing all loops and their maximal number of iterations in the application to the ISPTAP tool. This file is similar to the flow facts that are used by OTAWA [7]. It would also be possible to automatically extract the maximum iteration count of loops by the analysis of the application as e.g. done in [11], but this is out of scope for the ISPTAP tool.

2.2 Pipeline Execution Cost Analysis

2.2.1 Pipeline Execution Cost Analysis for CarCore Processor

The CarCore processor [35, 37] was developed in the CAR-SoC project as an simultaneous multi-threaded hard real-time capable processor. It is inspired by the Infineon TriCore processor [47]. The TriCore is a high-performance embedded processor for automotive applications, it combines the characteristics of a RISC, CISC, and a DSP processor. Since the instruction set of the TriCore supports more than 700 instructions, the CarCore implements only the subset of the instructions that is generated by the Hightec GCC compiler [22] and Tasking compiler [3]. Anyhow, the number of instructions supported by the CarCore is with 433 rather high.

The CarCore features two four-staged pipelines, one for address calculations and one for integer arithmetics. The additional loop pipeline that the TriCore processor has is not implemented in the CarCore. The Figure 2 gives a deeper insight into the CarCore architecture. In contrast to the TriCore processor the CarCore processor is simultaneous multithreaded (SMT). The differences to the TriCore and implementation of the CarCore architecture are described in detail in [35, 37].

The issue stage of the CarCore processor selects which thread is fed into which pipeline. It uses a fixed priority scheme for the scheduling of the different threads. One thread is able to be issued into both pipelines within one cycle, if an integer pipeline instruction is followed by an address pipeline instruction. The priority list of the issue stage is controlled by the second scheduling layer, that allows to apply different real-time scheduling schemes as [36, 38] proposes. The CarCore base architecture has separated instruction and data memories. Also an additional thread control block (TCB) memory is used. It holds information about the threads and is used as

Table 1: Instruction timing for the CarCore

Instruction	**Processing Time**
Address instruction	1 cycle
Integer instruction	1 cycle
Address instruction (as direct successor of an integer instruction)	0 cycles
Branches	3 cycles
Load (`LD`)	MMAT in cycles
Store (`ST`)	(MMAT - 1) cycles
Call microcode	58 cycles
Return microcode	51 cycles
`LDMST` microcode (Load-Modify-Store)	`LD`+1+`ST` cycles
`SWAP` microcode (Load-Modify-Store)	`LD`+`ST`+1 cycles
`ST.DA` microcode (Store Double-Word)	2 · `ST` cycles
`ST.T` microcode Store Bit	`LD`+1+`ST` cycles
Division is implemented in TriCore ISA by a sequence of `DVINIT`, `DVSTEP` and `DVADJ`, so no microcode is necessary.	

an interface to the second scheduling level. The CarCore does not support instruction prefetching or branch prediction to keep the timing model of the processor simple.

CarCore Instruction Execution Timing Model As described the CarCore processor features two pipelines, one for integer instructions and one for address instructions. Two instructions can be issued within one cycle, if an address instruction directly follows an integer instruction. Otherwise only one instruction can be issued to one of both pipelines. Complex instructions are implemented in interruptible microcodes consisting of micro-ops [37]. The Table 1 shows the cycle count needed to process the different instructions in the pipeline. The value 1 denotes that every cycle one instruction can be issued, 0 represents the case that the instruction is issued in parallel with another instruction. Values higher than 1 denote either microcodes with multiple micro-ops or instructions with additional latency times due to memory access or branch penalty. Notice that in addition to the processing time for the instructions shown in the Table 1, the latency needed to pass all pipeline stages after issuing is missing. This additional latency needs to be considered only once on the very last basic block of the application.

For *jumps* a branch penalty is added as extra latency to the execution time. It is needed to determine if a jump is taken or not and to deliver the calculated the target address to the fetch stage. This latency is fixed, because the CarCore does not support branch prediction. Therefore, the penalty has also to be charged for not taken branches to hinder the speculative execution before the branch direction is determined by the execute stage. Thus for any branch instruction 3 cycles are accounted.

If no data memory hierarchy is assumed, *loads* and *stores* are always directed to the off-chip memory. The support of any data memory hierarchy requires an additional address analysis for each memory access. Because the focus of the analysis is on the instruction memory, a support for data memory hierarchy is currently not implemented in the ISPTAP tool.

The processing time of the *load* instruction is given by the Maximum Memory Access Time (MMAT). This is due to the split-phase character of the memory instructions [37]: On issuing the first part of the *load* the pipeline calculates the address and requests the data from memory. The

issue stage has to delay the second part of the *load* that writes the loaded value into a register until the value is delivered, which will happen exactly after the memory latency number of cycles. Then the second part of the *load* arrives at the same cycle in the register write back stage as the memory controller delivers the loaded value. For *stores* the second part of the split-phase instruction is not needed, but the issue stage blocks dispatching further instruction until the *store* is completed. Because the *store* instruction does not need to write register values, the cycle in which the second part of the split-phase instruction would be placed is used by the next instruction. Therefore, the processing time for the *store* instruction is one cycle less than for a *load*.

Based on this timing model the ISPTAP tool is able to determine the execution cost of the instructions contained in a basic block for the CarCore processor. The timing effect between two basic blocks, e.g. if an integer instruction that is at the end of one basic block is the followed by an address instruction, which can be issued within the same cycle, is not addressed by the pipeline analysis. These timing effects have an impact on the overall WCET, but by using pessimistic bounds the proposed timing model does not underestimate those effects. Unfortunately, this will affect the tightness of the WCET estimate. Nevertheless, the timing analysis delivers a safe upper bound of the WCET. For a validation and a discussion of the tightness of the pipeline analysis refer to Section 2.6 and [30].

CarCore Fetch Timing Model Beside the basic block execution cost an additional cost that is caused by fetching the instructions of the basic blocks from memory has to be considered. Depending on the instruction memory latency this additional fetch cost affects the overall basic block cost with different impact.

Modelling the instruction fetch is done by calculating the ready time for each instruction. The ready time of an instruction within a basic block depends on the following parameters: the fetch block width, the fetch latency, the distance from the beginning of the basic block, the alignment of the instruction (it is possible that one instruction is stored in two fetch blocks), the instruction window size, and the fill state of the instruction window on entering the basic block.

The fetch block width and the fetch latency determine how many instructions can be fetched per cycle. By also taking the position of the instructions within the basic block into account the ready time of every instruction within a basic block can be calculated. If dynamic memories are used in the system that is analysed, the fetch cost determination assumes that every access will be a hit in the dynamic memory. The cost that is imposed by a miss will be added during the memory cost analysis independently in the next analysis step.

To determine the alignment of the instructions in the instruction memory it is necessary to use the same executable for analysis that will later be used by the system. The correct alignment is of importance, because only small changes of the code alignment, e.g. within a heavily used loop, may cause to a completely different timing behaviour.

The knowledge of the instruction window (IW) size is also important for a precise fetch timing model, because the fetch process is independent of the instruction issuing and execution. The fetch process requests instructions until the IW is filled, then the fetch process is stalled. It is restarted, if the pipeline issues an instruction and an IW entry is cleared. As described in [37] the IW of the CarCore holds 3 64 bit fetch blocks, which contain at least four instructions in sum[1]. Beside the size of the IW, its fill state on entering a basic block is of importance for a precise fetch timing analysis. Therefore, the fetch cost determination distinguishes the case, if a basic block is entered by *jumping* to it or by *continuous addressing*. In the first case the IW will be empty, because CarCore flushes the IW on any taken jump. Allowing jumps to instructions that are already in the IW without flushing the IW, would require a more elaborate analysis of the IW fill state. Therefore, the CarCore does not support this feature and always flushes the IW on every jump.

If a basic block is entered by exiting the preceding basic block without jumping (also denoted as *continuous addressing*), the ISPTAP tool is aware of the actual IW fill state left by the preceding block. So the instructions that are contained in the IW on entering the basic block are immediately

[1] Due to alignments to 16 bit addresses four 32 bit instructions can require three 64 bit aligned fetch blocks.

ready and their issuing is not delayed. For any further instructions in the basic block the ready time is calculated as described above.

Support for Simultaneous Multithreading The CarCore processor supports simultaneous multithreading and allows the execution of multiple threads with different priorities in parallel. The issuing policy of the CarCore ensures that the highest priority thread will be executed as if no other threads would run. Therefore, the timing model for the ISPTAP tool can ignore any effects of lower priority threads when analysing the highest priority thread. To provide a tight timing analysis of any lower priority thread the analysis would need to know the state of any higher priority tread, which is practically infeasible. On the other hand assuming always the worst-case behaviour of the higher priority threads would result in an infinite estimate, because it is possible that the highest priority thread uses the complete resources: e.g. it requires the complete fetch bandwidth or issues one instruction per cycle into each pipeline. For that reason the timing analysis in ISPTAP is only possible for the highest priority thread.

2.2.2 Pipeline Execution Cost Analysis for ARM Cortex M0 Processor

In this section the timing model of the ARM Cortex M0 with the ARMv6-M ISA [4] is shortly described.

ARM Cortex M0 Instruction Execution Timing Model The ARM Cortex M0 is a simple single issue in-order processor with a 3-stage pipeline. Therefore, the execution of an instruction is not influenced by any previous or subsequent instructions. So, the timing of the instruction execution can be determined easily. The pipeline timing model features different latencies for different classes of instructions as described in [5]. Table 2 gives an overview of the execution times for the different instruction types considered in the timing model. Notice that the instruction's execution times can be adjusted using the architectural configuration (see Section 3.2).

ARM Cortex M0 Fetch Timing Model The ARM Cortex M0 is currently modelled with an instruction window (IW) containing a single 64 bit fetch word, i.e. the IW can contain up to 4 16 bit instructions and fetches are performed with a 8 B granularity. During the pipeline analysis of a basic block the IW state of the preceding basic block is taken into account, if the basic block is not entered by a jump. Such that the IW content of the leaving basic block is used as initial IW state for its succeeding basic block. On jump the IW is considered as flushed. Notice that due to the minimal IW size of one fetch block, the IW is not filled autonomously causing the pipeline to stall, when the IW gets empty during a fetch request.

To model the fetch timing more precisely a careful examination of the timing for an implementation of the ARMv6-M architecture is required, e.g. using the FPGA model of the ARM Cortex M0. This is considered for future work.

2.3 Instruction Memory Cost Analysis

The instruction memory cost analysis is significantly different for static memories like scratchpads and dynamic memories as caches. For static memories an assignment of the instructions for the memory has to be found, which is considered in the timing of basic blocks. The quality of the resulting WCET estimate depends on the used memory assignment algorithm. For further information on scratchpad assignment algorithms the reader is referred e.g. to [50, 49, 52, 46, 15]. The analysis of dynamic memories requires the knowledge of their (possible) content during the execution of the application, which highly depends on the content management (i.e. replacement policy) used by the memory. To analyse caches different approaches for the common replacement policies are at hand. Further information regarding cache analysis are for example provided in [27, 26, 16, 39, 43, 19].

ISPTAP uses common analysis methods for scratchpads and caches, but also allows the analysis of the function-based D–ISP (refer to [30] for the details of the D–ISP analysis). To charge the

Table 2: Instruction timing for the ARM Cortex M0 according to [5]. The latency of loads and stores depends on the chosen memory latency.

Instruction	**Processing Time**
Arithmetic / Logical	1 cycle
Arithmetic / Logical with PC Register	3 cycles
Multiply	32 cycles
Read/Write Special Register	4 cycles
Branch Conditional - Taken - Not Taken	 3 cycles 1 cycle
Branch Unconditional	3 cycles
Branch Link	4 cycles
Branch Exchange	3 cycles
Branch Link & Exchange	3 cycles
Push / Pop (with n Registers)	1+n cycles
Pop and Return (with n Registers)	4 + n cycles
Load	MMAT cycles
Store	MMAT cycles
Data Synchronization Barrier	4 cycles
Data Memory Barrier	4 cycles
Instruction Synchronization Barrier	4 cycles

costs of the instruction memory system ISPTAP adds memory penalties for basic blocks, if during the execution of the basic block a cache or scratchpad miss occurs. The instruction memories ISPTAP supports are briefly described in the following.

2.3.1 Memory Cost Analysis and Optimization for Static Instruction Memories

Using static instruction memories the assignment of the code to the scratchpad memory is fixed. Hence, it is only necessary to determine which instruction, basic block, or function is in which memory and what is the memory latency of this memory. In the ISPTAP tool the following static memories are distinguished:

- **No first level memory (NO–MEM)**: All instructions are located in an off-chip memory.
- **First level memory only (S–ISP)**: All instructions are located in an on-chip scratchpad and are accessed with minimal latency.
- **Function-based static instruction scratchpad (FS–ISP)**: Selected functions are located in an on-chip scratchpad and can be accessed with minimal latency. The remaining part of the application is located in a slower off-chip memory.
- **Basic-block-based static instruction scratchpad (BBS–ISP)**: Selected basic blocks are located in an on-chip scratchpad and can be accessed with minimal latency. The remaining part of the application is located in a slower off-chip memory.

For the first two memories the memory cost analysis is implicitly done via pipeline execution cost analysis (described in Section 2.2) with the appropriate fetch bandwidth used in the fetch timing model.

The memories that allow a static assignment (FS–ISP and BBS–ISP) come in two flavours in the ISPTAP: a knapsack-based assignment (similar to [50], but using the WCET as metric) and the WCP-sensitive[2] assignment [46, 15]. For both approaches the ISPTAP needs to perform the pipeline cost analysis for the case that the code is in the scratchpad (S–ISP) and for the case that the code is in the off-chip memory (NO–MEM). Then the benefit to assign parts of the code to the scratchpad can be calculated by the memory cost analysis. The knapsack-based assignment requires a WCET analysis including the determination of the WCP before the code can be assigned to the scratchpad. This is not the case for the WCP-sensitive assignment, which is aware of the complete application control flow while finding the best assignment. For the detailed description of the used scratchpad assignment algorithms refer to [30].

Using the **FS–ISP** several functions are assigned to an on-chip scratchpad. The assignment of the functions is performed by solving the linear programs. After determining the set of assigned functions, the memory analysis considers for the execution of the selected functions the scratchpad memory access latency. For all other functions the latency for the access to the off-chip memory is taken into account. With this memory timing information from the memory cost analysis the ISPTAP tool calculates the final WCET estimate for a system with FS–ISP.

The approach for the **BBS–ISP** is similar as for the FS–ISP with two differences: the assignment is not done on the granularity of functions and the relocated basic blocks have to be reconnected to preserve the application's control flow. Basic blocks that enter or leave the scratchpad need to be altered by adding an additional jump or changing the jump target [15, 30]. Therefore, the different penalties for the additional jumps or changed jump targets need to be taken into account on calculating the WCET considering a BBS–ISP assignment. Furthermore, due to adding additional jumps the pipeline execution cost of basic blocks that are not altered can be affected: A basic block that was entered by continuous addressing and for which a connecting jump was added at the preceding basic block will now be entered by a jump, causing that the IW will be empty at the beginning of its execution. This results in a larger execution cost for this basic block that has to be considered by ISPTAP. Also the alignment of the basic blocks in the BBS–ISP needs to be respected during assignment to the scratchpad memory. Otherwise an undesired timing behaviour caused by changed alignment could arise. Therefore, it has to be assured that the alignment of the basic blocks copied to the BBS–ISP is kept or the alignment change has to be taken into account by recalculating the pipeline execution cost of the affected basic blocks. For simplicity, the ISPTAP tool assumes that the alignment of a basic block will not change by moving it into the BBS–ISP.

So the ISPTAP tool determines the assignment of the FS–ISP and BBS–ISP, models the timing effects of the scratchpad usage, and delivers a WCET estimate of a system with static instruction scratchpad memories. The foundations for the implementation of the WCET-aware knapsack-based code assignment for static memories were build by Sepp [45].

2.3.2 Memory Cost Analysis for Dynamic Instruction Memories

The analysis of dynamic memories requires a data flow analysis to keep track of the different contents of the memories during program execution. To improve the results of the analysis the graph on which the analysis is done can be extended by inlining functions and unrolling loops. Therefore, the ISPTAP uses the VIVU approach [29] to enhance the applications *supergraph*. This extended graph will be used for the memory analysis to determine the content of the dynamic memory on all paths of the applications. The following dynamic memories are modelled in the ISPTAP tool:

- **Fully associative instruction cache with LRU replacement (I–Cache$_{LRU}$)**: The approach proposed in [16] is used for this analysis.

[2]WCP – worst-case execution time critical path

- **Fully associative instruction cache with FIFO replacement (I–Cache$_{FIFO}$)**: The analysis tracks the all possible concrete cache states. Thus this analysis will not scale and is only implemented for comparison. For scalable and more precise FIFO cache analysis refer to [42, 18, 19].
- **Direct mapped instruction cache (I–Cache$_{DM}$)**: The approach proposed in [2] is used for this analysis.
- **Dynamic instruction scratchpad with LRU replacement (D–ISP$_{LRU}$)**: The analysis uses the method proposed in [30].
- **Dynamic instruction scratchpad with FIFO replacement (D–ISP$_{FIFO}$)**: The analysis uses a data flow analysis tracking all possible memory states as described in [30].
- **Dynamic instruction scratchpad with stack-based replacement (D–ISP$_{STACK}$)**: The analysis uses a data flow analysis tracking all possible memory states as described in [30].

The LRU and FIFO cache analysis supports only fully associative caches. Thus the results provided by ISPTAP are not affected by the separation of the instructions into the different sets. However, an analysis of set associative caches can be implemented by analysing multiple fully associative cache sets independently as e.g. described by Ferdinand and Wilhelm [17, 16]. The ISPTAP tool allows the selection of arbitrary values for the cache line size and the number of cache lines, although these numbers are typically only powers of two when implementing the caches in hardware. The analysis of configurations that cannot be efficiently implemented in hardware allows a better comparability of the cache memories and the scratchpad memories, since then the choice of the memory sizes in an evaluation does not depend on any restrictions of a hardware implementation. Beside the different replacement policies the design parameters of the D–ISP are the size of the scratchpad, the number of entries in the management tables, and the block size of the scratchpad memory.

After data flow analysis, the obtained memory penalty caused by cache or D–ISP misses is assigned to the basic blocks. For the D–ISP these memory penalties can occur only on calling a and returning to a function, whereas for caches a memory penalty may be assigned to every basic block.

2.4 Off-Chip Memory Cost

For simplicity the ISPTAP tool assumes for any off-chip memory access a constant latency. This assumption holds for SRAMs. In contrast to SRAMs the memory latency of DRAMs depends on the state of the memory controller: The DRAM memory is organised in rows and columns [23]. To access a certain word in the DRAM the addressed row has to be *activated*, which triggers the copying of the selected row into a row buffer. Then the memory word can be accessed via column *reads* or *writes*. To obtain multiple memory words a *burst access* can read/write multiple columns within the active row. To access a memory word in another row the active row has to be *precharged*, i.e. the content of the row buffer is written back into the memory. Beside the timing of the memory access in a DRAM additional latency caused by refreshing the memory has to be assumed. This is because the DRAM consist of capacitors, that lose their charge due to leakage current. So the bit information is lost, if the memory cells are not periodically refreshed. For a more detailed DRAM model including the different delays see [23].

Therefore, the memory access latency in DRAMs is not constant, as Bhat and Mueller [9] show for a real-world example. The ISPTAP tool can model DRAM accesses by assigning a constant safe upper bound for the DRAM access latency, assuming that the full memory access cycle including a refresh delay is charged for every single memory access. This leads to an overestimation of the memory latency, because row access delays for *activation* and *precharge* are considered for every memory access.

Table 3: Benchmarks for validation of the CarCore timing model (* including benchmark function and depending on the benchmark a harness and/or an initialisation function)

Benchmark	LOC	Structured Code	Loops	Nested Loops	Indirect Jumps	Function Count*
Adpcm	879	✓	✓	–	–	17
Bsort100	128	✓	✓	✓	–	3
Cover	240	✓	✓	–	✓	4
Crc	128	✓	✓	–	–	3
Duff	86	–	✓	–	✓	3
Fdct	239	✓	✓	–	–	2
Fibcall	72	✓	✓	–	–	2
Fir	276	✓	✓	✓	–	2
Matmult	163	✓	✓	✓	–	6
Nsichneu	4,253	✓	✓	–	–	1

To improve the preciseness of the ISPTAP tool regarding DRAM memory accesses it could apply one of the following optimisations. It would be possible to use a custom predictable memory controller that gives timing and bandwidth guarantees, as e.g. proposed by Akesson, Goossens, and Ringhofer [1]. Using common DRAMs the overhead caused by the periodic refresh can be considered in the WCET analysis for example by the approaches of Atanassov and Puschner [6] or Bhat and Mueller [9]. It is also possible to model the behaviour of the DRAM by a low-level analysis, as Bourgade et al. [10] propose.

2.5 WCET Estimation

Using the structural representation of the application, the timing cost of the execution of each basic block in the pipeline, and the additional memory penalties (either caused by static or dynamic memories) the calculation of a WCET estimate is possible. Puschner and Schedl [41] show how for a control flow graph the WCET can be calculated by using the IPET approach. The ISPTAP tool also employs the IPET approach. For the generation of the ILP formulation it uses the VIVU translated supergraph enriched by loop bounds, execution cost, and memory penalty. To find the maximal value of the objective function the ILP solving tool `lp_solve` is used [8]. It is also employed to find the assignments for the static memories.

2.6 Validation of the Timing Model

This section provides a short validation of the timing models implemented by the ISPTAP for several small benchmarks. Namely, the CarCore and the ARM Cortex M0 are supported at the current state.

2.6.1 Benchmarks

We selected a set of 10 micro-benchmarks from the Mälardalen benchmark suite [20] for the validation of the timing model. A short overview of the characteristics of the benchmarks is given in Table 3. The characterisation is partly obtained from [28]. Because the CarCore is multi-threaded, the usage of global variables were forbidden to ensure a thread-safe execution. Therefore, all global variables in the benchmark code are transferred into a structure that is thread-local. Furthermore, the main benchmark function was embedded in an thread harness to separate benchmark independent initialisation code. This was not necessary for the ARM Cortex M0 that can execute the benchmarks without any changes.

For the CarCore the benchmarks were executed by the cycle-accurate SystemC simulator of the CarCore processor [35] without any on-chip instruction memory. The memory access time for instruction and data memory is the same and is assumed as 4 cycles. For the ARM Cortex

Table 4: Overestimation of the CarCore timing by ISPTAP (The estimated execution time is compared to the execution time determined with the CarCore simulator for the same path.)

Benchmark	**Estimate (in Cycles)**	**Overestimation**	**Basic Blocks on Path**
Adpcm	1,057,720	19.3%	26,326
Bsort100	7,423	7.7%	405
Cover	8,510	16.1%	670
Crc	124,901	24.6%	6,406
Duff	3,050	31.7%	152
Fdct	3,531	2.8%	21
Fibcall	912	13.0%	21
Fir	5,422	9.3%	275
Matmult	428,108	10.8%	11,334
Nsichneu	12,394	4.7%	1,252

M0 timing validation a custom simulator (cf. [31]) that supports the ARMv6-M architecture and simulates the ARM Cortex M0 pipeline was used. For the memory system it is assumed that all instructions and data are located in an on-chip memory with memory access time of 1 cycle. After simulation the executed path of each benchmark was extracted and fed into the ISPTAP tool by creating adequate flow constraints. Thus the analysis performed by ISPTAP estimates the cost in cycles of the same path that was also executed by the simulator.

2.6.2 Validation of the CarCore Timing Model

The results of the analysis of the CarCore by ISPTAP are provided in Table 4. The third column of the table shows the differences of the estimates to the number of cycles needed to execute the benchmark on the CarCore processor simulator. This difference quantifies the overestimation of the analysis. As the shown in the table the overestimation ranges from very small values, like for Fdct with 2.8%, to a rather large impreciseness, as for Duff with 31.7%. Because the overestimation is a relative measure that depends on the cycle count needed to execute the considered path, the total estimate calculated by ISPTAP is shown in the second column the table. Furthermore, the number of basic blocks is provided to give insight in the length of the analysed path.

In average the overestimation of ISPTAP is about 14% for the CarCore processor without any on-chip memory (i.e. off-chip memory only). The timing analysis used the very same path as was executed by the CarCore SystemC simulator for each benchmark. Therefore, the overestimation cannot be caused by differences in the path assumed by ISPTAP and executed by the simulator. For a more detailed discussion of the reasons for the impreciseness of the CarCore timing model consult [30].

2.6.3 Validation of the Preliminary ARM Cortex M0 Timing Model

The validation results of the ARM Cortex M0 are shown in Table 5. In average the overestimation of ISPTAP is below 1%. The minimal overestimation is reached for Bsort100, Fdct, and Nsichneu which is below 0.1%. For Crc the largest overestimation of 2.7% has to be accounted. Since for the memory access time a minimal value (1 cycle) is chosen, the fetch timing using the single entry IW does not influence the analysis results. The timing model of the ARM Cortex M0 is preliminary and needs to be refined to reach more precise estimates. Anyhow, due to the simplicity of the processor pipeline the precision of the analysis is considered as sufficient. To further tune the timing model it is necessary to consider a physical implementation of an ARM Cortex M0 core as reference instead of the simulator.

Table 5: Overestimation of the ARM Cortex M0 timing by ISPTAP (The estimated execution time is compared to the execution time determined with the ARMv6-M architecture simulator for the same path.)

Benchmark	**Estimate (in Cycles)**	**Overestimation**	**Basic Blocks on Path**
Adpcm	1,435,355	0.8%	305,659
Bsort100	96,890	≈0%	20,303
Cover	2,834	0.4%	727
Crc	38,786	2.7%	7,432
Duff	1,251	0.2%	151
Fdct	8,941	≈0%	35
Fibcall	562	1.4%	91
Fir	11,338	0.1%	699
Matmult	754,251	0.8%	63,353
Nsichneu	10,054	≈0%	1,265

2.7 Limitations and Future Work

The reason why the overestimation for ISPTAP is higher depends mainly on the simpler pipeline timing model, that takes no interactions between different basic blocks into account, i.e. the timing of every basic block is considered as independent of any previous or subsequent basic blocks. Thus to prevent underestimating this effect, the timing model of the ISPTAP is more pessimistic, which leads to higher overestimations. This holds especially for the CarCore, which features a dual-issue core with an address and an integer pipeline.

In [25] the pipeline is modelled using *parametrized execution graphs*. This allows to take the effect on the timing of a certain basic block caused by basic blocks that are executed before and after into account. In [44], which described the usage of parametrised execution graphs to reduce the overestimation for basic blocks in out-of-order processors, it is stated that the execution timing of a basic block can increased or reduced by prefix/prologue and suffix/epilogue basic blocks/instructions. This is even possible for in-order superscalar processors, as the CarCore processor.

Another timing effect that is beyond the scope of one basic block is caused by the initial content of the instruction window (IW) on begin of the execution of a basic block. The content of the IW is crucial for a tight estimation of a basic blocks execution cost, because if an instruction is not contained the time to fetch the instruction is to be charged. Usually the IW already contains some instructions of the basic block on begin of its execution, if these instructions were fetched during the execution of the predecessor, i.e. the basic block is activated by continuous addressing. Anyhow, if a basic block is entered by a jump, the IW is usually flushed and is empty on execution start of the basic block. So the distinguishing of the execution context of a certain basic block is important for its execution cost. The ISPTAP tool can only distinguish, if a basic block is always entered by continuous addressing or not. In the first case it assumes the IW content is available that was left by its predecessor. In the second case ISPTAP timing model has to assume that the IW is empty, because it cannot distinguish the type of activation of the basic block, since this is dependent on the prior execution context (e.g. the basic block may entered by a jump or by continuous addressing, which is not distinguished by ISPTAP). So it is assumed that those basic blocks are always entered by a jump, resulting in an empty IW on the begin of their execution. This causes an overestimation, because the IW content has to be provided by fetches, which are possibly not necessary. Notice that this effect does not occur for the ARM Cortex M0 timing validation of Section 2.6.3, since a minimal fetch latency of 0 cycles is assumed. For the CarCore, which uses a memory access time of 4 cycles, this effect impacts the preciseness of the calculated estimate. Anyhow, ISPTAP delivers for each basic block a safe upper bound of its execution cost.

Since the focus of the ISPTAP tool is on the analysis and comparison of the WCET impact

of different instruction memories the level of overestimation shown in this section is considered as acceptable. To improve the results of the ISPTAP tool it is possible to tighten the pipeline timing model, especially for the CarCore, by employing one of the analysis techniques discussed above. For example execution graphs can be used to reduce the overestimation by taking previous basic blocks into account [44]. This was already done for the CarCore processor in the work of Landet [25], which reached an overestimation of in average slightly below 4% for a similar set of benchmarks from Mälardalen suite. To improve the pipeline timing model the pipeline execution cost analysis that is performed for each basic block separately has to be enhanced to model also inter-basic-block timing effects. By the structure of ISPTAP the additional software development effort to integrate another pipeline timing model is limited.

A further cause for overestimation is that context dependent flow constraints like loop bounds cannot be modelled in ISPTAP. Because no such content dependent flows occur in the analysed benchmarks above, no impreciseness can be charged to the flow constraints. However, ISPTAP supports only static flow information and thus only context independent loop bounds. So for loops with a variable maximum iteration count that depend on the context of the loop the maximum possible loop bound has to be considered for all contexts. For example for a nested loop in which the iteration count of the inner loop depends on the iteration of the outer loop ISPTAP has to use the maximum possible iteration count for every execution context of the inner loop. Thus the execution cost of the nested loop will be overestimated. To support context dependent flow constraints the capabilities of ISPTAP have to be extended to import flow facts for different execution contexts and assign them to the correct position in the supergraph and to the flow variable in the IPET formulation of the application. The issues of the formulation of proper context dependent flow information and the mapping of these information to the flow constraints in an IPET formulation are addressed by Engblom and Ermedahl [13]. Therefore, Engblom and Ermedahl introduce the concept of *scoped flow information* that allows to model a wide range of flow descriptions. An implementation of this concept, or any similar that is as powerful, requires changes in ISPTAP that are restricted to the processing of the flow facts and the enrichment of the graph that is used to build the ILP with the flow information. Anyhow, such enhancements are also left for future work.

Further, the support of on-chip data memories like caches [24, 17] and scratchpads [46, 12, 51], which requires the analysis of the addresses accessed by load/store instructions, and off-chip memories as SDRAM [10] is considered as possible extension to ISPTAP.

3 Usage & Configuration

ISPTAP uses for the description of the analysis to perform a configuration file with the options defined in Section 3.1. This configuration employs two further configuration files one for defining the timing model of the used architecture, which is described in Section 3.2, and one to set up the log output level, discussed in Section 3.3.

The Figure 3 gives an overview of the required input files (I1 - I5) and the potential output files (O1 - O7) of an analysis run. The in-/output files are described below:

I1: Analysis Config The analysis configuration file (`.cfg`) contains the main analysis settings and defines all other input files.

I2: Application Memory Dump The memory dump file (`.dump`) to be analysed. It is set by the option `dump_file` in the analysis configuration file.

I3: Flow Facts The flow facts (`.ff`) of the program under analysis, which is set by the option `flow_fact_file` in the analysis configuration file.

I4: Architectural Config The configuration file of the used architectural timing model (`.prop`), which is set by the option `architecture_config_file` in the analysis configuration file.

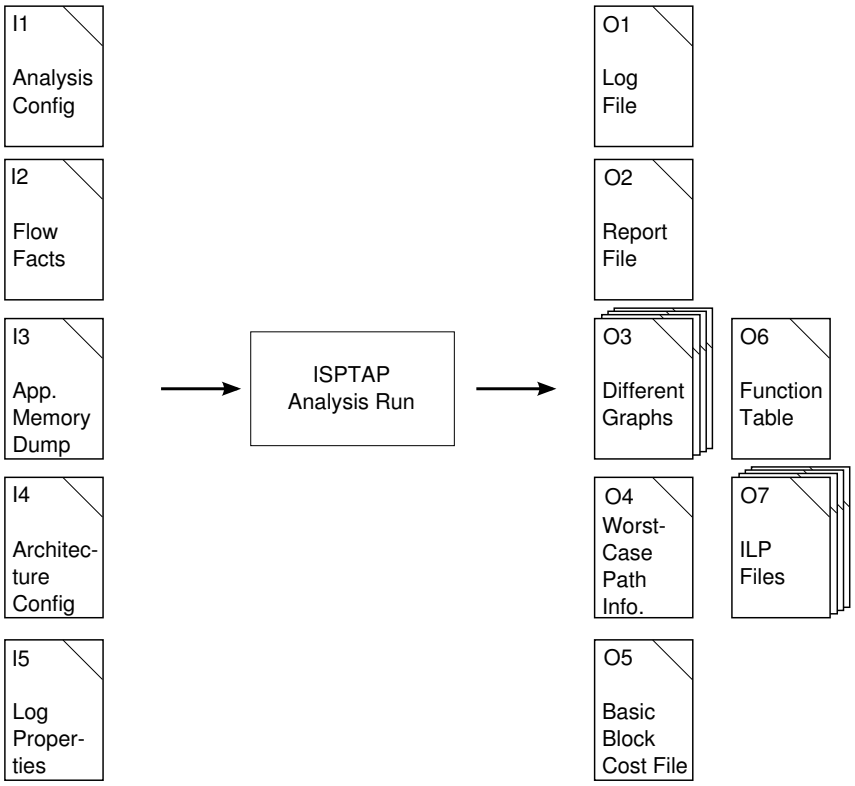


Figure 3: In- and Output Files of ISPTAP

I5: Log Properties The property file for the log4cxx logging facility (`.prop`) that defines the log level. The log property file is set by the option `log_properties_file` in the analysis configuration file.

O1: Log File The main log output file (`.log`) containing all output specified by the log level that is set in the log property file. The log file name is defined by the option `log_file` in the analysis configuration file.

O2: Report File The report file (`.rpt`) contains only the analysis results, e.g. the estimated WCET. The report file name is set by the option `report_file` in the analysis configuration file.

O3: Different Graphs Different control flow and call graphs either in Graphviz or GraphML format (as defined by `export_format`). The creation of the different graphs is activated by the setting of the following options in the analysis configuration file:
`export_function_cfgs`, `export_function_call_graph`, `export_scfg`, `export_flow_scfg`, `export_solved_flow_scfg`, and `export_solved_flow_scfg_with_assignment`.

O4: Worst-Case Path Information These output files (`.wcp` and `.wch`) contain worst-case path information as basic block address trace or as basic block address histogram. The creation of the files is activated by the options `export_wcpath` and `export_wcpath_hist` and the file name prefix is set by `wcpath_fileprefix` in the analysis configuration file.

O5: Basic Block Cost File A file (`.cst`) containing the estimated WCETs for each basic block. The basic block cost output is configured via the analysis configuration file. The creation

of the file is activated by `export_bb_cost_from_graph` and the file name prefix is set by `export_bb_cost_fileprefix`.

O6: Function Table A table (`.ftab`) containing information about all parsed functions in the application. The function table output is configured in the in the analysis configuration file. The function table file name is set by the option `function_table_file` and its creation is activated by `export_function_table`.

O7: ILP Files Files (`.ilp`) for each generated ILP that is solved during analysis. The generation of these files is activated by the option `export_ilps` in the analysis configuration file.

In the following the required configuration files are described and most important configuration parameters are listed. In all configuration files comments are allowed at the beginning of a line using either `//` or `#`. Additionally, command line parameters can be set to override certain options of the configuration files, which is useful for batch processing. For further information regarding the configuration of ISPTAP and command line parameters, use the `--help` option of ISPTAP.

3.1 Analysis Configuration

The analysis configuration file sets up the necessary parameters for an analysis including the input file, the entry function, the used architecture, the log output file, and also the graphs to export. An assignment of a configuration is done as follows: `isptap.property = value`. Notice that in the following description of the parameters the `isptap` prefix is omitted.

3.1.1 Program to be analysed

`dump_file` ***(default: --none--):*** The tool analysis memory dump files, which can be generated from ELF-files by using `objdump`. This parameter defines the memory dump file to be analysed.

`entry_function` ***(default: main):*** The top function that is to be analysed. All functions called by this function are included into the analysis.

`use_flow_fact_files` ***(default: 0):*** If set to '1', the analysis uses a flow-fact file providing information about loop bounds, other flow constraints, and indirect jump or call targets.

`use_flow_fact_graph_enrichment` ***(default: 0):*** If set to '1', the information of the flow-fact file will be added to the graph representation of the program. This is necessary to respect the flow information in the analysis of the program.

`flow_fact_file` ***(default: --none--):*** The name and the relative path to the flow-fact file containing additional flow information of the program, including loop bounds, absolute flow information, and indirect jump and call targets. For help on the format and usage of the flow-fact file read the help message by using the command line parameter `--help-flow-facts`.

3.1.2 Metrics

`use_metric` ***(default: WCET):*** The used analysis metric. The default metric is the worst-case execution time in which the path with the longest execution time of the program is determined. Other metrics are MDIC, which selects the path of the program with the Maximum Dynamic Instruction Count, and MPL (Maximum Path Length), which selects the longest path of the program (w.r.t. the number of instructions). The latter metrics do not apply a processor timing model and are added just for debug reasons.

3.1.3 Architecture

`architecture` ***(default: CARCORE):*** Specifies the architecture model that is used for timing analysis. Supported architectures are: `CARCORE` and `ARMV6M`.

`use_architecture_config_file` ***(default: 1):*** If set to '1', an architectural configuration file is used. Otherwise a default architecture is assumed.

`architecture_config_file` ***(default: configs/carcore.prop):*** The file that configures the architecture that is analysed. For further information see Section 3.2.

3.1.4 Memory configuration

`memory_type` ***(default: NONE):*** Sets the used instruction memory type. Supported are the following memory types:

- Static instruction scratchpads with basic block assignment policy: `BBSISP` (Knapsack algorithm), `BBSISP_JP` (Knapsack algorithm with jump and size penalties), `BBSISP_WCP` (WCET path sensitive algorithm, without jump and size penalties), and `BBSISP_JP_WCP` (WCET path sensitive algorithm with jump and size penalties, according to [15]).
- Static instruction scratchpads with function assignment policy: `FSISP` (Knapsack algorithm) and `FSISP_WCP` (WCET path sensitive algorithm)
- Instruction caches: `ICACHE` with different replacement policies, see `memory_replacement_policy`.
- Dynamic instruction scratchpad: `DISP` with different replacement policies, see `memory_replacement_policy`.
- No on-chip instruction memory: `NONE` (all fetches are handled by the off-chip memory).

`memory_size` ***(default: 0):*** Size of the memory in bytes.

`memory_start_size` ***(default: 0):*** Start memory size in bytes for the memory range analysis mode (see `memory_size_stepping`).

`memory_step_size` ***(default: 0):*** Size of the steps in bytes used in memory range analysis mode (see `memory_size_stepping`).

`memory_size_stepping` ***(default: 0):*** If set to '1', multiple analyses with a range of memory sizes are performed for the chosen memory type and application. The analysis starts with the size defined in `memory_start_size` and increases in steps defined in `memory_step_size` until the value defined in `memory_size` is reached.

`memory_cache_BBs` ***(default: 0):*** Sets the cache line granularity to basic blocks (i.e. a basic block fits always a cache line and a cache line contains always only one basic block).

`memory_bbsisp_add_jump_penalties_to_wcet` ***(default: 0):*** This option is used for BBS–ISP flavours that do not consider any penalties for additional jumps on assignment of basic blocks to the scratchpad (i.e. `BBSISP` and `BBSISP_WCP`). If the parameter is set to '1', the penalties for connecting the chosen basic blocks by additional jumps are added before the final WCET calculation. Thus, the provided WCET will be correct, concerning the necessary jump penalties, but these penalties were not considered during the finding of the scratchpad assignments.

`memory_disp_ignore_outsized_functions` ***(default: 0):*** If set to '1', the D–ISP ignores functions that are larger than its memory size. In that case these functions will be fetched from the off-chip memory.

`memory_replacement_policy` ***(default: UNKNOWN):*** The replacement policy of the dynamic memories, D–ISP and cache. Supported policies are: `FIFO`, `LRU`, `DIRECT_MAPPED` (cache only), and `STACK` (D–ISP only).

`istpap.bbsisp_wcp_shrink_ilp_formulation` ***(default: 0):*** Set this option to '1', if the ILPs of `BBSISP_WCP` and `BBSISP_JP_WCP` (which is human readable) are too complex to be solved quickly. Then the WCP-sensitive ILP formulation used to find the basic block assignment will be simplified. This is done by inlining simple constraints, like the edge costs. Notice that the provided scratchpad assignment is not affected by this option.

`bbsisp_wcp_fill_isp_up` ***(default: 0):*** This option allows the BBS–ISP assignment algorithms to add basic blocks to the BBS–ISP, even if they have no positive impact on the WCET. This is done by adding the used scratchpad size to objective function weighted with `1e-10`, which fills the scratchpad as much as possible. Notice due to the weighting of the used scratchpad size the assignment of useful bocks shall not be affected.

3.1.5 Output configuration

`log_file` ***(default: isptap_default.log):*** The file name of the output log file.

`log_properties_file` ***(default: configs/baselog.prop):*** The used log4cxx log property file, which defines the log level.

`report_file` ***(default: report.log):*** The file name of the report file.

`report_append` ***(default: 0):*** If set to '1', the report file is not overwritten and the report is appended.

`static_mapping_report` ***(default: 0):*** If set to '1', the functions that are assigned for the FSISP are exported. Also header files that assign the selected functions to the SPM section are generated for the analysed application.

`static_mapping_report_file` ***(default: report_smapping.cfg):*** The file name prefix for the reports and headers of the FSISP assignment.

`function_table_file` ***(default: isptap_functiontable.ftab):*** The file name of the function table to export.

`old_function_table_format` ***(default: 0):*** If set to '1', the old function table format (that contains only name and start address) is used for export.

`export_format` ***(default: GRAPHVIZ):*** Output format of graphs: `GRAPHVIZ` or `GRAPHML`.

`export_function_cfgs` ***(default: 0):*** If set to '1', the control flow graphs of all functions are exported.

`export_function_call_graph` ***(default: 0):*** If set to '1', the call graph (of the entry function) is exported.

`export_scfg` ***(default: 0):*** If set to '1', the super control flow graph (CFG of the entry function in which all calls are inlined) is exported.

`export_flow_scfg` ***(default: 0):*** If set to '1', the super control flow graph enriched with the flow facts is exported.

`export_solved_flow_scfg` ***(default: 0):*** If set to '1', the super control flow graph with the calculated flows of the worst-case path is exported.

`export_solved_flow_scfg_with_assignment` ***(default: 0):*** If set to '1', the super control flow graph with the calculated flows of the worst-case path is exported. Additionally, the code (basic blocks) that is assigned to the scratchpad (BBSISP or FSISP) is highlighted.

`export_function_table` ***(default: 0):*** If set to '1', a function table containing the name, start address, and size of each function is exported.

`export_wcpath` ***(default: 0):*** If set to '1', a basic block address trace of the WCET critical path is exported.

`export_wcpath_hist` ***(default: 0):*** If set to '1', a basic block histogram of the WCET critical path is exported.

`wcpath_fileprefix` ***(default: isptap_wcpath):*** The prefix for the file names for the exported WCET critical path statistics.

`export_wcpath_instr_stats` ***(default: 1):*** If set to '1', the statistics of the different instruction types on the WCET critical path are exported.

`export_bb_cost_from_graph` ***(default: 0):*** If set to '1', the cost and activation count of every basic block on the WCET critical path is exported.

`export_bb_cost_fileprefix` ***(default: isptap_bbcost):*** The file prefix of the basic block cost file.

`export_ilps` ***(default: 0):*** If set to '1', the generated ILPs needed to calculate the estimate are exported.

3.2 Architectural Timing Model Configuration

To model the architecture, on which the application under analysis should be executed, a configurable architectural timing model of the supported CarCore and ARM Cortex M0 processor is used. To parametrise the timing model an architecture configuration file can be used. Depending on the architecture (set by `isptap.architecture` in the analysis configuration file) the configuration file has different properties, including latencies of instructions or memory access times. In the following the different parameters are discussed for the supported architectures.

The architectural configuration file is defined by the options `use_architecture_config_file` and `architecture_config_file` in the analysis configuration file. If no architectural configuration is provided, default values for the specified architecture are used.

3.2.1 CarCore

The timing model of the CarCore was developed for the evaluation of the D–ISP in [30] comparing it to other on-chip instruction memories. Therefore, it provides several properties that allows it to adjust the instruction memory configuration and the fetch process.

The architectural configuration file for the CarCore supports the following properties. The configuration file uses the property file format. An assignment of a configuration is done as follows: `carcore.property = value`. Notice that in the following description of the parameters the `carcore` prefix is omitted. Comments are allowed using either `//` or `#`.

`branch_latency` ***(default: 3):*** The latency of branches (in cycles).

`call_latency` ***(default: 57):*** The latency of the call microcode (in cycles).

`return_latency` ***(default: 50):*** The latency of the return microcode (in cycles).

`fetch_independent_imem` ***(default: 1):*** If set to '1', the instruction and data memory are independent of each other and use a separated memory connection.

`use_fetch_optimization_branch_ahead` ***(default: 0):*** If set to '1', the processor stalls the instruction fetch, if a branch instruction is in the instruction window. This optimisation of the CarCore is currently not supported.

`use_fetch_optimization_enough_instrs` ***(default: 0):*** If set to '1', the processor analyses the number of instructions in the instruction window and stops fetching when enough instructions are present. This optimisation of the CarCore is currently not supported.

`fp_latency_arithmetic` ***(default: 1):*** The latency of arithmetic floating point instructions (in cycles).

`fp_latency_divide` ***(default: 5):*** The latency of division floating point instructions (in cycles).

`fp_latency_conversion` ***(default: 1):*** The latency for conversion floating point instructions (in cycles).

`fp_latency_multiplyaccumulate` ***(default: 4):*** The latency for multiply-accumulate floating point instructions (in cycles).

`fp_latency_multiply` ***(default: 3):*** The latency for the multiplication floating point instruction (in cycles).

`fp_latency_sqrtseed` ***(default: 10):*** The latency for the square root seed floating point instruction (in cycles).

`fp_latency_updateflags` ***(default: 0):*** The latency for the update flags floating point instruction (in cycles).

3.2.2 ARMv6-M

ISPTAP was extended to support the ARM Cortex M0 processor (implementing the ARMv6-M architecture), which is a simple and small processor. By its simplicity of the architecture the timing model of the ARM Cortex M0 is less complex than the CarCore. Thus, it allows preciser analyses with less analysis effort. Currently, the timing model of the ARM Cortex M0 does not support on-chip instruction memories, but we plan to integrate the analyses that were performed for the CarCore processor.

The architectural configuration file for the ARM Cortex M0 supports the following properties. The configuration file uses the property file format. An assignment of a configuration is done as follows: `armv6m.property = value`. Notice that in the following description of the parameters the `armv6m` prefix is omitted. Comments are allowed using either `//` or `#`.

`arithmetic` ***(default: 0):*** The latency of arithmetic instructions (in cycles).

`arithmetic_with_pc` ***(default: 2):*** The latency of arithmetic instructions using the PC register (in cycles).

`logic` ***(default: 0):*** The latency of logic instructions (in cycles).

`multiply` ***(default: 31):*** The latency of multiply instructions (in cycles).

`read_special_reg` ***(default: 3):*** The latency of the *read special register* instruction (in cycles).

`write_special_reg` ***(default: 3):*** The latency of the *write special register* instruction (in cycles).

`branch_conditional_taken_latency` ***(default: 2):*** The branch latency of conditional branches when taken (in cycles).

`branch_conditional_nottaken_latency` ***(default: 0):*** The branch latency of conditional branches when not taken (in cycles).

`branch_unconditional_latency` ***(default: 2):*** The latency of branch instructions (in cycles).

`branch_and_link_latency` ***(default: 3):*** The latency of branch and link instructions (in cycles).

`branch_and_exchange_latency` ***(default: 2):*** The latency of branch and exchange instructions (in cycles).

`branch_link_and_exchange_latency` ***(default: 2):*** The latency of branch, link and exchange instructions (in cycles).

`pop_and_return_extra_latency` ***(default: 3):*** The additional latency for the (returning) jump, if the PC is in the register list of pop instructions (in cycles)

`data_sync_barrier` ***(default: 3):*** The latency of the *data synchronisation barrier* instruction (in cycles).

`data_mem_barrier` ***(default: 3):*** The latency of the *data memory barrier* instruction (in cycles).

`instr_sync_barrier` ***(default: 3):*** The latency of the *instruction synchronisation barrier* instruction (in cycles).

3.2.3 Memory Configuration

The memory configuration defines the memory subsystem of the architecture's timing model. The configuration properties are supported in the architectural configuration files of both architectures: CarCore and ARM Cortex M0. Depending on the used architecture the appropriate prefix (`carcore` or `armv6m`) has to be used for the property assignment in the architectural configuration file.

`load_latency_onchip` ***(default: 0):*** The latency of loads for on-chip memories (in cycles).

`load_latency_offchip` ***(default: 4):*** The latency of loads for off-chip memories (in cycles).

`store_latency_onchip` ***(default: 0):*** The latency of stores for on-chip memories (in cycles).

`store_latency_offchip` ***(default: 3):*** The latency of stores for off-chip memories (in cycles).

`fetch_latency_onchip` ***(default: 0):*** The latency of fetches for on-chip memories (in cycles).

`fetch_latency_offchip` ***(default: 3):*** The latency of fetches for off-chip memories (in cycles).

`fetch_bandwidth` ***(default: CarCore:64 / ARMv6-M:32):*** The fetch bandwidth of the instruction memory (in bit).

`jump_penalty_for_continuous_addressing` ***(default: 5):*** The CA jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by continuous addressing need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_jump_disp4` ***(default: 0):*** The J4 jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by a short jump (disp4) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_jump_disp8` ***(default: 0):*** The J8 jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by a short jump (disp8) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_jump_disp15` ***(default: 0):*** The J15 jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by a jump (disp15) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_jump_disp24` ***(default: 0):*** The J24 jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by a jump (disp24) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_jump_ind` ***(default: 0):*** The JI jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by an indirect jump need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_call_disp8` ***(default: 0):*** The C8 jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by a short call (disp8) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_call_disp24` ***(default: 0):*** The C24 jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by a call (disp24) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`jump_penalty_for_call_ind` ***(default: 0):*** The CI jump penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in cycles) is used in the case that two basic blocks that were connected by an indirect call need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_continuous_addressing` ***(default: 4):*** The CA basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by continuous addressing need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_jump_disp4` ***(default: 2):*** The J4 basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by a short jump (disp4) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_jump_disp8` ***(default: 2):*** The J8 basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by a short jump (disp8) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_jump_disp15` ***(default: 0):*** The J15 basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by a jump (disp15) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_jump_disp24` ***(default: 0):*** The J24 basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by a jump (disp24) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_jump_ind` ***(default: 0):*** The JI basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by an indirect jump need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_call_disp8` ***(default: 2):*** The C8 basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by a short call (disp8) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_call_disp24` ***(default: 0):*** The C24 basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by a call (disp24) need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`size_penalty_for_call_ind` ***(default: 0):*** The CI basic block size penalty for the BBSISP assignment algorithm (BBSISP_JP and BBSISP_JP_WCP). The penalty (in bytes) is used in the case that two basic blocks that were connected by an indirect call need to be reconnected when one of the basic blocks is to be moved to the static scratchpad.

`address_width` ***(default: 32):*** The address width the processor (in bit).

`disp_block_size` ***(default: 8):*** The size of the memory blocks in the DISP (in bytes).

`disp_block_load_latency` ***(default: 0):*** The latency for loading one memory block into the DISP (in cycles).

`disp_controller_hit_cycles` ***(default: 4):*** The number of cycles the DISP controller needs for function hit handling.

`disp_controller_miss_cycles` ***(default: 4):*** The number of cycles the DISP controller needs for function miss handling.

`disp_max_function_size` ***(default: 2^{22}):*** The maximum size of a function in the DISP (in bytes).

`disp_mapping_table_size` ***(default: 256):*** The maximum number of entries of the DISP's mapping table.

`disp_lookup_width` ***(default: 256):*** The number of mapping table entries the DISP controller looks up in one cycle (i.e. the associativity of the mapping table)

`disp_context_stack_depth` ***(default: 16):*** The maximum allowed context stack depth (or maximum call depth) that defines the size of the DISP's context stack memory.

`icache_line_size` ***(default: 32):*** The line size of the instruction cache (in bytes). (This parameter will be ignored, if `memory_cache_BBs` set.)

`icache_associativity` ***(default: --none--):*** The associativity of the cache. Use '1' for direct-mapped and '0' for fully-associative.

`icache_miss_latency` ***(default: 0):*** The latency of a cache miss (in cycles).

3.3 Log Output Configuration

ISPTAP uses the log4cxx-library for logging. The level of the log output is configured by a log property file, which can be set by the `isptap.log_properties_file` parameter in the analysis configuration file.

4 License and Availability

The ISPTAP tool is licensed under GPL Version 3. The source code is available at `https://github.com/smetzlaff/isptap`.

Acknowledgement

The author likes to thank Jörg Mische for the supplement of necessary CarCore instruction decoding code and the support on modelling the CarCore architecture.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer. "Predator: a Predictable SDRAM Memory Controller". In: *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS '07)*. New York, NY, USA: ACM, Oct. 2007, pp. 251–256. DOI: 10.1145/1289816.1289877.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. "Cache behavior prediction by abstract interpretation". In: *Proceedings of the Third International Symposium on Static Analysis (SAS '96)*. Vol. 1145. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Sept. 1996, pp. 52–66. DOI: 10.1007/3-540-61739-6_33.

[3] Altium. *TASKING VX-toolset for PCP User Guide*. v3.2. Altium. Feb. 2009.

[4] ARM Limited. *ARMv6-M Architecture Reference Manual*. Tech. rep. ARM DDI 0419C (ID092410). Sept. 2010.

[5] ARM Limited. *Cortex -M0 – Technical Reference Manual*. Tech. rep. Revision: r0p0. ARM DDI 0432C (ID113009). Nov. 2009.

[6] P. Atanassov and P. Puschner. "Impact of DRAM Refresh on the Execution Time of Real-Time Tasks". In: *Proceedings of the IEEE International Workshop on Application of Reliable Computing and Communication*. Dec. 2001, pp. 29–34.

[7] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *Proceedings of 8th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*. Vol. 6399. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Oct. 2011, pp. 35–46. DOI: 10.1007/978-3-642-16256-5_6.

[8] M. Berkelaar, K. Eikland, and P. Notebaert. *Open source (Mixed-Integer) Linear Programming system*. http://lpsolve.sourceforge.net/5.5/. Version: 5.5.0.13 Released: 08/05/2008 [Online, last accessed on 3rd March 2012]. Aug. 2008.

[9] B. Bhat and F. Mueller. "Making DRAM Refresh Predictable". In: *Proceedings of 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2010, pp. 145–154. DOI: 10.1109/ECRTS.2010.23.

[10] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. "Accurate analysis of memory latencies for WCET estimation". In: *Proceedings of the 16th International Conference on Real-Time and Network Systems (RTNS 2008)*. Oct. 2008.

[11] M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. "Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation". In: *Proceedings of 14th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA '08)*. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2008, pp. 161–166. DOI: 10.1109/RTCSA.2008.53.

[12] J.-F. Deverge and I. Puaut. "WCET-Directed Dynamic Scratchpad Memory Allocation of Data". In: *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS '07)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2007, pp. 179–190. DOI: 10.1109/ECRTS.2007.37.

[13] J. Engblom and A. Ermedahl. "Modeling complex flows for worst-case execution time analysis". In: *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2000, pp. 163–174. DOI: 10.1109/REAL.2000.896006.

[14] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. "Worst-case execution-time analysis for embedded real-time systems". In: *International Journal on Software Tools for Technology Transfer (STTT)* 4 (4 Aug. 2003), pp. 437–455. DOI: 10.1007/s100090100054.

[15] H. Falk and J. Kleinsorge. “Optimal static WCET-aware scratchpad allocation of program code”. In: *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC '09)*. New York, NY, USA: ACM, July 2009, pp. 732–737. DOI: 10.1145/1629911.1630101.

[16] C. Ferdinand and R. Wilhelm. “Efficient and Precise Cache Behavior Prediction for Real-Time Systems”. In: *Real-Time Systems* 17 (2 Nov. 1999), pp. 131–181. DOI: 10.1023/A:1008186323068.

[17] C. Ferdinand and R. Wilhelm. “On predicting data cache behavior for real-time systems”. In: *In Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*. Vol. 1474. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, June 1998, pp. 16–30. DOI: 10.1007/BFb0057777.

[18] D. Grund and J. Reineke. “Abstract Interpretation of FIFO Replacement”. In: *Proceedings of the 16th International Symposium Static Analysis (SAS 2009)*. Vol. 5673. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Aug. 2009, pp. 120–136. DOI: 10.1007/978-3-642-03237-0_10.

[19] D. Grund and J. Reineke. “Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection”. In: *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2010, pp. 155–164. DOI: 10.1109/ECRTS.2010.8.

[20] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. “The Mälardalen WCET Benchmarks: Past, Present And Future”. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, July 2010, pp. 136–146. DOI: 10.4230/OASIcs.WCET.2010.136.

[21] R. Heckmann and C. Ferdinand. *Worst-Case Execution Time Prediction by Static Program Analysis*. Tech. rep. [Online, last accessed on 3rd March 2012]. AbsInt Angewandte Informatik GmbH, 2006. URL: http://www.absint.de/aiT_WCET.pdf.

[22] HighTec. *HighTec EDV-Systeme GmbH Website*. http://www.hightec-rt.com/. [Online, last accessed on 3rd March 2012].

[23] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007. ISBN: 978-0-12-379751-3.

[24] S.-K. Kim, S. L. Min, and R. Ha. “Efficient worst case timing analysis of data caching”. In: *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. Los Alamitos, CA, USA: IEEE Computer Society, June 1996, pp. 230–240. DOI: 10.1109/RTTAS.1996.509540.

[25] C. Landet. “Worst Case Execution Time Evaluation for a Simultaneous Multithreaded Processor”. In: *Proceedings of 2nd Junior Researcher Workshop on Real-Time Computing*. Rennes, France: IRISA, Oct. 2008, pp. 9–12.

[26] Y.-T. S. Li, S. Malik, and A. Wolfe. “Efficient microarchitecture modeling and path analysis for real-time software”. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 1995, pp. 298–307. DOI: 10.1109/REAL.1995.495219.

[27] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. “An accurate worst case timing analysis for RISC processors”. In: *IEEE Transactions on Software Engineering* 21.7 (July 1995), pp. 593–604. DOI: 10.1109/32.392980.

[28] Mälardalen Real-Time Research Center (MRTC). *WCET Benchmark Suite*. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. [Online, last accessed on 3rd March 2012].

[29] F. M. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. “Analysis of Loops”. In: *Proceedings of the 7th International Conference on Compiler Construction, (CC ’98)*. Vol. 1383. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Apr. 1998, pp. 80–94. DOI: 10.1007/BFb0026424.

[30] S. Metzlaff. “Analysable Instruction Memories for Hard Real-Time Systems”. PhD thesis. Universität Augsburg, July 2012.

[31] S. Metzlaff, J. Mische, and T. Ungerer. “A Real-Time Capable Many-Core Model”. In: *Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session*. Nov. 2011, pp. 21–24.

[32] S. Metzlaff and T. Ungerer. “Replacement Policies for a Function-based Instruction Memory: A Quantification of the Impact on Hardware Complexity and WCET Estimates”. In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2012, pp. 112–121. DOI: 10.1109/ECRTS.2012.22.

[33] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. “A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware”. In: *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS 2011)*. Vol. 6566. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Feb. 2011, pp. 122–134. DOI: 10.1007/978-3-642-19137-4_11.

[34] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. “Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors”. In: *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA ’08)*. New York, NY, USA: ACM, Oct. 2008, pp. 38–45. DOI: 10.1145/1509084.1509090.

[35] J. Mische. “Echtzeitfähige Ablaufplanung für simultan mehrfädige Prozessoren”. PhD thesis. Universität Augsburg, Dec. 2011.

[36] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. “Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads”. In: *Proceedings of the 26th IEEE International Conference on Computer Design 2008 (ICCD ’08)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2008, pp. 371–376. DOI: 10.1109/ICCD.2008.4751887.

[37] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer. “How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT”. In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS 2010)*. Vol. 5974. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Feb. 2010, pp. 2–14. DOI: 10.1007/978-3-642-11950-7_2.

[38] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. “IPC Control for Multiple Real-Time Threads on an In-Order SMT Processor”. In: *Proceedings of the Fourth International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2009)*. Vol. 5409. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Jan. 2009, pp. 125–139. DOI: 10.1007/978-3-540-92990-1_11.

[39] F. Mueller. “Timing Analysis for Instruction Caches”. In: *Real-Time Systems* 18 (2 May 2000), pp. 217–247. DOI: 10.1023/A:1008145215849.

[40] E. M. Myers. “A precise inter-procedural data flow algorithm”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’81)*. New York, NY, USA: ACM, Jan. 1981, pp. 219–230. DOI: 10.1145/567532.567556.

[41] P. P. Puschner and A. V. Schedl. “Computing Maximum Task Execution Times – A Graph-Based Approach”. In: *Real-Time Systems* 13.1 (July 1997), pp. 67–91. DOI: 10.1023/A:1007905003094.

[42] J. Reineke and D. Grund. “Relative competitive analysis of cache replacement policies”. In: *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES ’08)*. New York, NY, USA: ACM, June 2008, pp. 51–60. DOI: 10.1145/1375657.1375665.

[43] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. "Timing Predictability of Cache Replacement Policies". In: *Real-Time Systems* 37.2 (Nov. 2007), pp. 99–122. DOI: 10.1007/s11241-007-9032-3.

[44] C. Rochange and P. Sainrat. "A Context-Parameterized Model for Static Analysis of Execution Times". In: *Transactions on High-Performance Embedded Architectures and Compilers II*. Lecture Notes in Computer Science 5470 (2009), pp. 222–241. DOI: 10.1007/978-3-642-00904-4_12.

[45] A. Sepp. "WCET-basierte Optimierung der Befehlsabbildung für On-Chip-Speicher". Diploma Thesis. Universität Augsburg, 2009.

[46] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. "WCET Centric Data Allocation to Scratchpad Memory". In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2005, pp. 223–232. DOI: 10.1109/RTSS.2005.45.

[47] TriCore. *TriCore 1 User's Manual - Volume 1 Core Architecture*. V1.3.8. Infineon Technologies AG. Nov. 2007.

[48] TriCore. *TriCore 1 User's Manual - Volume 2 Instruction Set*. V1.3.8. Infineon Technologies AG. Nov. 2007.

[49] L. Wehmeyer and P. Marwedel. *Fast, Efficient and Predictable Memory Accesses: Optimization Algorithms for Memory Architecture Aware Compilation*. Springer, 2006. ISBN: 1-4020-4821-1.

[50] L. Wehmeyer and P. Marwedel. "Influence of Onchip Scratchpad Memories on WCET Prediction". In: *Proceedings of the 4th International Workshop on Worst-Case Execution Time Analysis (WCET 2004)*. Rennes, France: IRISA, June 2004, pp. 29–32.

[51] J. Whitham and N. Audsley. "Implementing time-predictable load and store operations". In: *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09)*. New York, NY, USA: ACM, Oct. 2009, pp. 265–274. DOI: 10.1145/1629335.1629371.

[52] J. Whitham and N. Audsley. "Using Trace Scratchpads to Reduce Execution Times in Predictable Real-Time Architectures". In: *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2008, pp. 305–316. DOI: 10.1109/RTAS.2008.11.